On Upgrading the Numerics in Combustion Chemistry Codes

DOUGLAS A. SCHWER, JOHN E. TOLSMA, WILLIAM H. GREEN, JR., and PAUL I. BARTON*

Department of Chemical Engineering, Massachusetts Institute of Technology, 77 Massachusetts Ave., Cambridge, MA 02139, USA

A method of updating and reusing legacy FORTRAN codes for combustion simulations is presented using the DAEPACK software package. The procedure is demonstrated on two codes that come with the CHEMKIN-II package, CONP and SENKIN, for the constant-pressure batch reactor simulation. Using DAEPACK generated code, analytical derivative calculations, sparsity pattern information, and hidden discontinuity information can be obtained for the models of interest. This information can be easily integrated with different solvers giving the modeler great flexibility in selecting the best solution procedure. Using the generated code, the CONP code was connected to three different solvers, and the SENKIN code was connected to two different solvers. The effect of model formulation, analytical derivatives, sparsity, and sensitivity equation solution method were analyzed for three large kinetic mechanisms for methane, acetylene, and n-heptane. For the n-heptane model, with 544 species and 2446 reactions, a factor of 10-speed improvement over the original solution procedure was found using analytical derivatives and sparse linear algebra. For sensitivity calculations, for a small number of parameters, a factor of 55 improvement over the original solution procedure was found for the n-heptane problem. Upon closer examination of results, no one method is found to always be superior to other methods, and selection of the appropriate solution procedure requires an examination of the specific kinetic mechanism, which is easily conducted using DAEPACK generated code. © 2002 by The Combustion Institute

INTRODUCTION

During the past two decades, computer simulations have become an important tool for designing and investigating combustion systems. Considerable effort has been devoted to generate chemistry models applicable to a wide range of reaction conditions, as well as to more accurately predict the existence of pollutant species, which have concentrations several orders of magnitude smaller than the main reactants and products. Much progress has been made in the last decade towards this goal; however, to accomplish this the reaction mechanisms have increased greatly in complexity because of the addition of many radicals and other transient species into the mixture description.

The number of species and reactions involved in these new mechanisms is often staggering. One example of this is methane/air combustion, which is as basic as one can get for hydrocarbons, but requires anywhere from 30 to 50 species and hundreds of reactions to accurately describe the chemistry over a wide range of conditions. Complex fuels have also received much attention, and over the last five years several programs have been developed to build reaction mechanisms for different fuels based on well understood reaction rules [1]. For more complex fuels, the number of species in the model can easily be several hundred, with the number of reactions in the thousands.

Although the older computer codes can still handle simulations for these much more complex systems (sometimes with extensions and adjustments), they often use routines that become inefficient quickly as the number of species and reactions increases. There are two approaches to help minimize this. One method is to work on robust ways to reduce the size of the chemical mechanisms while retaining sufficient accuracy. There have been several efforts along these lines and much progress has been made [2–5], but efficient general algorithms for reducing mechanisms and ensuring the reduced models are accurate over the range of conditions important in a combustion simulation are still not available.

Another approach is to replace the existing solution procedures with newer procedures that take advantage of advances made in numerical analysis during the last decade. To do this,

* Corresponding author. E-mail: pib@mit.edu

0010-2180/02/$–see front matter
PII 0010-2180(01)00352-2

however, significant portions of these legacy codes must be rewritten to take advantage of the new solvers, which is often a time consuming and error prone task. Unless the interfaces remain consistent, a considerable amount of work is also necessary to validate the new code to the same degree as the previous, older codes.

In this paper, we present a new method to automate this process of updating legacy codes. We demonstrate this procedure on a simple chemical simulation and show how this can improve computational performance in some cases by over an order of magnitude. To accomplish this, we use a software package developed at MIT called DAEPACK [6]. We first describe DAEPACK and then discuss the chemical systems and solution techniques used for these simulations. We then apply DAEPACK to calculate analytical derivatives and sparsity information for three separate chemical kinetic mechanisms, and examine the efficiency of the derivative calculations. Finally, we compute the resulting simulations with and without sensitivity computations, using several solvers and techniques that can be easily interfaced with DAEPACK generated code. All computations were conducted on an 800 MHz Pentium-III computer with 256 MB running Linux.

DESCRIPTION OF DAEPACK

The oldest and most prevalent approach used for modeling and simulation is the FORTRAN-based paradigm. The model is typically encapsulated into a single subroutine that calculates the residuals for the system of equations of interest. Very often, the residual subroutine calls third-party subroutine libraries to compute the residuals. Older solution techniques typically require only the residual subroutine and some other easily obtainable model information. Newer techniques, however, often require much more specific information about the system of equations, but also offer much more efficient, robust, and sometimes more accurate solution procedures. This information is typically difficult to obtain and code into the subroutine, especially with the use of calls to third-party subroutine libraries which can have large, complex, and interconnected hierarchies of subroutines.

DAEPACK is a collection of programs developed to help modelers apply modern algorithms and techniques to their FORTRAN[1] models efficiently and accurately. At the core of DAEPACK is a source code analyzer. The analyzer is able to parse FORTRAN source code and extract information needed for more modern solvers, such as the calculation of analytical derivatives, sparsity information, and hidden discontinuities. DAEPACK then generates automatically auxiliary subroutines to calculate this information, making it available to the modeler for analysis and for interfacing with modern solvers.

For this paper, two aspects of DAEPACK are demonstrated for programs of interest to the combustion community. The first aspect is the generation of analytical derivatives. For stiff ODE/DAE solution procedures, derivatives of the model are necessary for using robust solution techniques. Numerical derivatives, especially for very large models, are costly and not always accurate, and determining and coding the analytical derivatives by hand can be time consuming for complex subroutine libraries. Several methods for obtaining analytical Jacobians automatically have been developed [7]. DAEPACK currently implements the sparse forward mode method of automatic differentiation [7] and we plan to add other modes in future releases of the software.

In addition to calculating analytical derivatives, DAEPACK can also generate sparsity patterns for a particular model. With the model in symbolic form, the generated code can automatically determine the sparsity pattern and adapt it to the current state of the model. This is extremely useful for kinetic models, where one program can generally read in many different kinetic models, each having its own sparsity pattern. The occurrence information can then be used by the modeler to estimate the potential of exploiting sparsity, and also as input into sparse linear algebra solvers such as the Harwell MA48 libraries [8].

[1]The current implemention is for FORTRAN code, however, the approach is valid for all procedural programming languages.

It is important to recognize that there is a fundamental difference between automatic differentiation (AD) and partial derivatives computed by symbolic computing environments such as Maple and Mathematica. On the one hand, AD generates a segment of code that can evaluate *values* for partial derivatives of the dependent variables at any given *values* for the independent variables. In particular, symbolic expressions for the partial derivatives are never constructed at any point. On the other hand, a symbolic computing environment will (automatically) derive symbolic expressions for the partial derivatives, and then encode these expressions as a subroutine to evaluate derivative values, if this is required. The disadvantage with this latter approach is that the symbolic expressions for partial derivatives can grow very rapidly in complexity, whereas the AD approach of only computing values remains a priori bounded in terms of memory usage and computational cost [7].

The AD component of DAEPACK is one of several software tools available for obtaining numerical values of derivatives of functions coded in imperative programming languages such as C, C^{++}, and Fortran. Several variants of AD exist [7], essentially differing by how the chain-rule is applied to the underlying elementary operations of the target code, however, most AD tools produce derivative code via one of two ways. The first is by using the operator overloading features of several modern programming languages, including C^{++} and Fortran-95. Operator overloading is an advanced programming feature that enables a user to redefine mathematical operators (e.g., +, −, ., and /) for user-defined data types. AD tools employing this technique provide special data types for selected program variables in the code. These data types carry with them the information necessary for computing derivative values. The user can redeclare certain program variables to be this data type and the compiler will generate automatically new instructions for computing the derivative values. Some tools employing this approach are ADOL-C [9], ADOL-F [10], BC1 [11], and GC1 [12]. The other main approach for producing derivative code is source-to-source transformation. Using compiler technologies, new source code for evaluating the derivative values is generated from the original source code evaluating the function of interest. The advantage of this approach is that portable code is generated. Also, the user does not need to have access to a compiler which supports operator overloading. In many of these source-to-source tools, rather than requiring the user to redeclare selected variables, the user must simply specify which of the program variables are independent and dependent variables. The AD tool will automatically identify which intermediate variables are directly involved in the computation of the derivatives. In the operator overloading approach, it is often the responsibility of the user to identify these *active* intermediate program variables manually. Some tools employing source-to-source transformation are JAKEF [13], GRESS [14], PADRE2 [15], ADIFOR [16], Odyssée [17], TAMC [18], and DAEPACK [6].

DAEPACK provides several options for computing derivative values. The option applied in this paper is very similar to the algorithm used in ADIFOR. Specifically, the *reverse mode* is applied at the statement level and the overall derivative values are propagated through the code with the *forward mode*. The implementation in DAEPACK has been designed from the beginning for efficient accumulation of sparse derivative matrices. In the sparse forward mode used in this paper, only nonzero derivative values are propagated and carefully optimized utility routines are provided to reduce the overheads associated with working only with nonzero entries. ADIFOR also provides a library, SparsLinC, for computing sparse derivative matrices. Another option available in DAEPACK for exploiting sparsity is to use seed matrix compression [7] in conjunction with the sparsity pattern information. However, basic row or column compression is unsuitable for the Jacobians considered in this paper because they all contain some dense rows and columns.

In addition to generating code for derivative evaluation and sparsity pattern determination, DAEPACK provides several components for generating other codes evaluating information necessary to perform advanced numerical calculations. For example, given a Fortran source code evaluating a model, DAEPACK provides a component that generates new code computing the natural interval extension of the model [19]. This new code may be used, for example, to solve systems of equations using interval Newton/generalized bisection. Another component generates new code computing the convex relaxations of nonconvex functions in the model [20]. This code is necessary for many global optimization and non-convex mixed integer non-linear programming algorithms. Finally, a component has been developed to generate the information necessary to perform reliable and correct numerical integration and parametric sensitivity analysis of models containing discontinuities [21–23]. This DAEPACK component generates new code that allows the user to *lock* the model so that it evaluates a smooth function. This is achieved by setting a flag that fixes the trace of statements executed despite the presence of IF statements and discontinuous intrinsic functions such as MIN and MAX. This new code also extracts the *discontinuity functions* associated with the discrete events, which are necessary for performing simulation and sensitivity analysis correctly and efficiently.

The current paper applies DAEPACK to two relatively simple programs used extensively by the combustion community and included with the CHEMKIN-II packages [24, 25]. Both programs simulate a constant-pressure batch reactor. The first program, CONP, computes only the simulation, while the second program, SENKIN [26], computes the simulation with sensitivities. We first construct the model and then outline the solution techniques used for solving the model. We then apply DAEPACK to the system and examine the sparsity of some common, large scale kinetic systems and the efficiency of analytical derivative calculations as compared to numerical differences. Then we implement new solvers into the CONP and SENKIN code and evaluate the performance of using DAEPACK and the new solvers with CONP and SENKIN.

GOVERNING EQUATIONS

The system we examine in detail solves the adiabatic, constant pressure problem for a perfectly stirred, batch reactor. This system was chosen for its simplicity, yet it has many things in common with more complex systems. The governing ordinary differential equations are given below:

$$\rho\frac{dY_i}{dt} = W_i w_i \tag{1}$$

$$\rho C_p\frac{dT}{dt} = -\sum_{k=1}^{N} W_k h_k w_k \tag{2}$$

where ρ is the gas density, T is the temperature, C_p is the mixture specific heat, Y_i is the mass fraction, W_i is the molecular weight, w_i is the net species production rate, and h_i is the enthalpy of species i. The code uses the CHEMKIN-II library to calculate both the thermodynamic properties as well as the chemical production and destruction rates. The basic chemical equations are summarized here, but the interested reader should consult the original CHEMKIN report for more details.

For this paper, we consider an elementary reaction system with N species and M reactions. In general, any reaction j is described by its stoichiometric coefficients:

$$\sum_{i=1}^{N} \nu'_{ij}A_i \rightleftharpoons \sum_{k=1}^{N} \nu''_{kj}A_k \tag{3}$$

where ν'_{ij} and ν''_{ij} are the stoichiometric coefficients for species i, reaction j. For elementary reactions, each reaction rate ν_j is calculated as:

$$r_j = k_{f,j}(T)\prod_{k=1}^{N} C_k^{\nu'_{kj}} - k_{b,j}(T)\prod_{k=1}^{N} C_k^{\nu''_{kj}} \tag{4}$$

where C_k is the concentration of species k, $k_{f,j}$ is the forward-rate temperature dependent term, and $k_{b,j}$ is the backward-rate temperature dependent term. For the majority of reactions, the forward-rate temperature dependent term $k_{f,j}$ is calculated using Arrhenius coefficients, $k_{f,j} = A_j T^{nj}\exp(-E_{a,j}/RT)$, where A_j is termed the A-factor for reaction j. The backward temperature dependence is either determined in a similar manner, or more often calculated through equilibrium arguments using the thermo-chemical data. The concentration of species k can be expressed in terms of the density:

$$C_k = \rho Y_k / W_k \tag{5}$$

and the density in turn is related to the species mass fraction by the ideal gas law:

$$\rho(P, T, Y_i) = \frac{P}{RT}\left(\sum_{i=1}^{N}\frac{Y_i}{W_i}\right)^{-1} \tag{6}$$

Third body reactions also play an important role in many of these reaction mechanisms. Third body reactions are written as:

$$r_j = k_{f,j}\,(T)C_{M,j}\prod_{k=1}^{N} C_k^{\nu'_{kj}} - k_{b,j}\,(T)C_{M,j}\prod_{k=1}^{N} C_k^{\nu''_{kj}} \tag{7}$$

where $C_{M,j}$ is the third body effective concentration, calculated as:

$$C_{M,j} = \sum_{i=1}^{N}\alpha_{ij}C_i \tag{8}$$

where α_{ij} is a third-body efficiency and varies for different reactions and different species. α_{ij} is generally one for most of the minor species but may vary for the major species. Because of this, the third body concentration is calculated in CHEMKIN as:

$$C_{M,j} = \sum_{i=1}^{N}C_i + \sum_{i=1}^{N}\beta_{ij}C_i \tag{9}$$

where $\beta_{ij} = \alpha_{ij} - 1$. In addition to these simple third body reactions, CHEMKIN-II provides means for calculating pressure dependent third body reactions, which tend to be complex non-linear functions whose descriptions are not required here. Please consult the CHEMKIN papers for a complete explanation and discussion of them.

The net molar production rate for any given species is then described in terms of the individual reaction rates and their associated stoichiometric coefficients:

$$w_i = \sum_{j=1}^{M}(\nu''_{ij} - \nu'_{ij})\, r_j \tag{10}$$

We investigate the constant-pressure batch reactor for three specific chemical kinetic mechanisms indicative of the type of chemical kinetic systems that are becoming more prevalent, examining calculations for both simple simulations, and simulations with sensitivities computed for the reaction A-factor parameters. All three systems are described within the CHEMKIN-II framework and are available on the internet.

The first mechanism we examine is the GRI-Mech 3.0 methane/air mechanism [27]. The GRI mechanism is a well documented system and has been used extensively for both spatially homogeneous and non-homogeneous [28, 29] systems. Version 3.0 of this mechanism has 53 species, 325 reactions, and contains extensive nitrogen chemistry. The specific case examined for this paper is a stoichiometric mixture of methane and air at one atmosphere and 1,500 K. We seed the simulation with a small amount of O and H radicals to start the simulation.

The second mechanism we investigate is an acetylene flame soot-formation mechanism from Wang and Frenklach [30]. This mechanism has 99 species and 533 reactions. Because the mechanism focuses on soot production, it includes chemistry that is considerably different than the nitrogen/methane chemistry in GRI-Mech. The specific case examined for this paper is an acetylene/air mixture with 7.75(mol/mol)% acetylene by volume at a temperature of 1,000 K and a pressure of one atmosphere. Again, the simulations are started with a small concentration of O and H radicals.

The final mechanism that we investigate is an n-heptane mechanism of Curran et al. [31]. With 544 species and 2446 reactions, this mechanism represents the type of large mechanism that will become more common in the future. The specific case examined for this paper is an n-heptane/air mixture with 0.14(mol/mol)% n-heptane at a pressure of 12.5 atm and 800 K.

Mechanisms such as these typically rely on both extensive databases of carefully measured reactions as well as the incorporation of reactions based on generalized reaction families and reaction rules. Extensive and careful analysis is required to determine the correctness and validity of large and complex mechanism such as these. Any enhancement to robustness and efficiency that can be obtained with the current infrastructure of codes would be a tremendous benefit to these analyses.

SOLUTION PROCEDURE

We first look at the solution of the constant pressure problem as described by Eqs. 1 and 2 without calculating sensitivities. For stiff problems such as most chemical kinetic problems, the most popular approach is a variable order, variable time-step BDF method originally attributed to Gear [32]. To use BDF methods, we first write the system in a general way,

$$F\,(\dot{Z}, Z, t, p) = 0 \tag{11}$$

subject to the initial conditions,

$$\phi\,(\dot{Z}\,(t_0), Z\,(t_0), t_0, p) = 0 \tag{12}$$

where Z is our vector of dependent variables, $\dot{Z}$ is the time derivative of our vector of dependent variables, t is our independent variable (in this case, time), and p are the parameters of the problem. This equation describes general differential-algebraic equation (DAE) systems; an ODE system is a subset of the above where

$$F\,(\dot{Z}, Z, t, p) = \mathbf{I}\dot{Z} - f\,(Z, t, p) = 0 \tag{13}$$

where $\mathbf{I}$ is the identity matrix. For an Nth order BDF method, we discretize $\dot{Z}$ as:

$$\dot{Z}_{n+1} = \sum_{i=0}^{N} \alpha_i Z_{(n+1)-i} \tag{14}$$

where n is the current time-step. The coefficients α_i are typically a function of both the order N and the time-step h of the computation, and are discussed in detail in [32, 33]. The full discretized equation that is solved by the BDF method is written as:

$$F\left(\sum_{i=0}^{N} \alpha_i Z_{(n+1)-i}, Z_{n+1}, t_{n+1}, p\right) = 0 \tag{15}$$

Using a Newton method to solve the above non-linear equation, we obtain the iterative equation:

$$Z_{n+1}^{(k+1)} = Z_{n+1}^{(k)} - \mathbf{J}^{-1} G\,(Z_{n+1}^{(k)}, Z_n, Z_{n-1}, \ldots, Z_{(n+1)-N}, t_{n+1}, p) \tag{16}$$

where G is the discretized form of F, and k is the iteration level. The Jacobian $\mathbf{J}$ is defined as:

$$\mathbf{J} = \alpha_0 \frac{\partial F}{\partial \dot{Z}} + \frac{\partial F}{\partial Z} \tag{17}$$

Note that the above expression simplifies for ODE systems to the following:

$$\mathbf{J} = \alpha_0 \mathbf{I} - \frac{\partial f}{\partial Z} = \alpha_0 \mathbf{I} - \mathbf{J}' \tag{18}$$

Extensive research has gone into creating efficient, robust BDF solvers with strict error control. Most popular methods work by minimizing the number of Jacobian evaluations and LU decompositions required, typically by reusing old Jacobians from previous time-steps and carefully monitoring the convergence rate of the quasi-Newton iteration to ensure good convergence. The heuristics for these programs are quite complex and require years of adjusting to obtain optimum and robust performance. For ODE systems, the VODE package [34] has obtained widespread acceptance. It uses up to a 5th-order BDF method with a variable time-stepping procedure to obtain results within specified error bounds. For general DAE systems, DASSL [33] has gained widespread acceptance, and similarly uses up to a 5th-order BDF method with a variable time-stepping procedure. For very large systems, even with the use of defered Jacobians, the Jacobian calculation and LU factorization are still the most costly parts of the simulation. Both of these solvers assume dense, meaning that most of the elements of $\mathbf{J}$ are non-zero, or banded linear systems. However, many of the large systems encountered in combustion simulations are extremely sparse but lack any sort of regular structure such as bands. For these systems, significant gains in efficiency can be made by exploiting this sparsity in the Jacobian evaluations and in the linear algebra using direct sparse Gauss elimination. DSL48S [35], based on DASSL, was written to take advantage of this form of sparsity by using the Harwell MA48 sparse linear algebra libraries [8]. For computations contained in this paper, we use VODE, DASSL, and DSL48S for our simulations.

Parametric sensitivities for the dynamic system described above can be found by solving auxiliary sensitivity equations along with the original system. For the ith parameter, the additional sensitivity equations are:

$$\frac{\partial F}{\partial \dot{Z}} \dot{s}_i + \frac{\partial F}{\partial Z} s_i + \frac{\partial F}{\partial p_i} = 0 \tag{19}$$

subject to the initial conditions,

$$\frac{\partial \phi}{\partial \dot{Z}} \dot{s}_i (t_0) + \frac{\partial \phi}{\partial Z} s_i (t_0) + \frac{\partial \phi}{\partial p_i} = 0 \tag{20}$$

where $s_i \equiv \frac{\partial Z}{\partial p_i}$.

For each parameter p_i, there are $N + 1$ sensitivity variables for the system defined by Eqs. 1 and 2, where N is the number of species. There are two important observations to note about the above sensitivity equations. The first observation is that clearly the original dynamical system is not coupled to the sensitivity equations, and can be solved independently of the sensitivity equations, although the sensitivity equations are dependent on the dynamical system. The second observation is that the sensitivity equations are linear in the sensitivity variables with a Jacobian matrix identical to that employed for the state equations.

Unlike the simulations, there are several competing methods for solving the sensitivity equations. In this paper we investigate three different methods. The first method is the staggered direct method [36, 37]. It divides each time step into two computations, first computing the original system, and then computing the sensitivity equations. Because the sensitivity equations are linear, this method directly inverts the system to solve the sensitivity equations. By doing a direct inversion, however, the method must factor the Jacobian at every step, which can become very expensive for large systems.

The second method is called the simultaneous corrector method, described by Maly and Petzold [38]. This solution procedure treats the system as one large non-linear dynamical system, and takes advantage of the resultant structure of this system to obtain an efficient method for solving the sensitivity equations. The advantage of this method is that the Jacobian does not need to be factored at every step.

The third method commonly used is the staggered corrector method, and is described in Feehery, Tolsma, and Barton [35]. This method, as in the staggered direct method, separates the solution procedure at each time step into two phases, solving the original system and then solving the sensitivity equations. Unlike the staggered direct method, however, the staggered corrector method solves the sensitivity equations iteratively using the deferred Jacobian. By using an iterative solution procedure, the Jacobian again is not required to be factored at every step, and can thus provide a substantial increase in performance.

The original SENKIN code uses the staggered direct method for computation, as implemented in DASAC [37]. For the computations reported in this paper, we use the DASPK3.0 [39] and DSL48S [35] libraries for the simulations with sensitivities. All three solvers are based on the DAE-solver DASSL. DASPK incorporates all three methods for solving sensitivity systems, and provides an enhancement to the DASAC implementation for ill-conditioned problems [39]. Like DASSL, it uses dense linear algebra for factorizations and back substitutions. DSL48S uses the staggered corrector method for computing sensitivities, and as mentioned previously uses the Harwell MA48 sparse linear algebra libraries.

All of these programs provide the ability to calculate numerical Jacobians or use a user supplied analytical Jacobian. For many systems, a user provided analytical Jacobian can make the solution procedure more robust, and can also increase the efficiency tremendously. DAEPACK provides a method for generating the necessary analytical Jacobians easily and integrating them with different solvers.

APPLICATION OF DAEPACK

For this paper, we use DAEPACK to upgrade two codes that come with the CHEMKIN-II library called CONP and SENKIN. CONP solves the constant pressure system described in

Eqs. 1 and 2 using the VODE integration package. The VODE package requires that the user encapsulates the function $f(Z, t, p)$ in Eq. 13 as an external subroutine, and pass it along with the dependent variables, time, and parameters to the VODE subroutine. For the CONP program, the vector of dependent variables is simply $Z = (Y_i, T)^T$. The functional form of f is easily determined from Eqs. 1 and 2 and is encapsulated into the subroutine FUN. Using CHEMKIN-II, the actual length of the subroutine FUN is quite small (20 lines), as most of the real computational work is done within the calls to CHEMKIN-II library subroutines (which remained unchanged throughout this work). SENKIN solves the same system as CONP, plus the sensitivty of the system to the reaction A-factors using the DASAC solver [37]. DASAC is based on the DASSL DAE solver and uses the staggered direct method for computing sensitivities. To use DAE based solvers we must compute the residual function $F(\dot{Z}, Z, t, p)$ as defined in Eq. 11, which is carried out in a subroutine called RES. Constructing RES from FUN is very simple. For this system, RES is again quite small, as most of the work is carried out in the CHEMKIN-II library subroutine.

The authors of CONP and SENKIN opted to use the finite-difference Jacobian generation subroutine that comes as part of VODE and DASSL to determine the Jacobian matrices $\mathbf{J}'$ and $\mathbf{J}$. To switch to analytical Jacobians, we first run the source code through the DAEPACK code analyzer. DAEPACK requires all of the main subroutines, and the libraries that the main subroutines access (in this case, the CHEMKIN-II libraries), be available as source code. DAEPACK then generates a subroutine that computes the necessary derivatives for the Jacobian using the sparse forward mode analytical Jacobian procedure, as well as the sparsity information needed by the Harwell MA48 subroutine. The user then writes a small wrapper subroutine to interface the generated code with the desired solver. Several examples of wrappers for commonly used solvers are included within the DAEPACK distribution.

Using sparsity information returned from the automatically generated code, we first examine the sparsity of the three chemical systems mentioned previously, given the original formulation in CONP and two alternative formulations for the same constant-pressure problem. Second, we examine the improvement in performance using analytical Jacobians with all three systems. Third, for the sparse systems, we examine using both analytical Jacobians and sparse linear algebra to improve performance using different solvers. Finally, we examine the effect of solution technique, sparsity, and analytical derivatives on the sensitivity calculations.

EVALUATION OF ANALYTICAL JACOBIANS

The destruction and formation of most chemical species is usually dominated by only a handful of reactions. Reflecting this physical reality, most large chemical kinetic models in the literature are extremely sparse in terms of interactions between different species. That is, physically each species reacts with at most only a small number of the total number of species which exist. Most solution methods in current use, however, rely on dense techniques and do not attempt to take advantage of this inherent sparsity. For this section, we use DAEPACK generated code to examine the sparsity of the actual numerical simulations, and determine ways to exploit this sparsity in numerical solvers.

For numerical solvers, the sparsity of the system is dependent on the non-zero entries in the Jacobian $\mathbf{J}$, as defined in Eq. 17. For the original formulation where $Z = (Y_i, T)^T$, inspection of the original conservation Eqs. 1 and 2 reveals that the Jacobian $\mathbf{J}$ is totally dense. The reason for this is that every concentration found in the rate Eq. 7 is dependent on every species mass fraction Y_i through Eqs. 5 and 6. This results in very poor performance for the current automatic analytical derivative techniques, and also for sparse linear algebra. In the resulting computations, we refer to this as the density formulation.

The choice of $Z = (Y_i, T)^T$ is a common one, but not the only one. An equivalent system can be constructed for the constant pressure problem by using the species moles n_i and the system volume V as dependent variables instead of species mass fractions, where $n_i = C_iV$. The species conservation equations are simply

$$\frac{dn_i}{dt} = w_i V \tag{21}$$

The differential equation for the volume can be derived from the ideal gas law $V = nRT/P$. For constant pressure, the differential equation for volume is

$$\frac{dV}{dt} = \frac{V}{n} \sum_{k=1}^{N} w_i V - \frac{V}{\rho C_p T} \sum_{k=1}^{N} W_k h_k w_k \tag{22}$$

For this case, the dependent variables become $Z = (n_i, T, V)^T$ and the system of differential equations is Eqs. 21–22 with the energy Eq. 2. The advantage of this system is that the Jacobian is now sparse, unlike our original formulation, and we have kept an ODE formulation. The disadvantage of this system is that both n_i and V are extensive properties, whereas the input of CONP is in mole-fractions, temperature, and pressure. Because of this, we have freedom in selecting the initial volume and total moles to anything satisfying $PV = nRT$. For the simulations presented here, we select the initial volume such that initially the species moles are equivalent to the species mole-fractions. Because the mole-fractions are the same order of magnitude as the mass-fractions, this allows us to use the same relative and absolute error tolerances that we used for the density simulations. In the subsequent computations, we refer to this as the mole formulation.

We can also rearrange the original system $Z = (Y_i, T)^T$ and add ρ to the dependent variable vector Z. Thus, $Z = (Y_i, T, \rho)^T$ becomes our vector of dependent variables; and the additional equation becomes:

$$\rho - \rho(P, T, Y_i) = 0 \tag{23}$$

where $\rho(P, T, Y_i)$ is computed from the ideal gas law, Eq. 6. The advantage of this system is that it solves the exact same equations as the original system, but the introduction of the additional dependent variable ρ now makes the Jacobian matrix $\mathbf{J}$ sparse. The system, however, is now a DAE system, and a DAE solver such as DASSL or DSL48S must be used to solve it, as discussed above. This is less of an issue when computing sensitivities, because the most popular programs for computing sensitivities (DASPK, DASAC, DSL48S) are all based on the DAE solver DASSL. In the resulting numerical calculations, we refer to this as the augmented density formulation.

Using the above formulations, we examined the sparsity for all three mechanisms mentioned previously. Although the mechanisms were fairly large, we found that the Jacobians for these mechanisms were still very dense, even for the sparse formulations. The GRI-Mech was 90% dense, while the acetylene flame was 63% dense. Only the n-heptane was significantly sparse, with 90% sparsity.

Upon inspection of the CHEMKIN-II libraries, we found that this was caused by third body reactions. Because the third body concentration was calculated through Eq. 9, any species involved as a reactant or product in a third-body reaction had a dense row for the Jacobian. This can be circumvented by using a slightly different method for calculating the third-body concentration. Calculating this concentration as:

$$C_{M,j} = C + \sum_{i=1}^{N} \beta_{ij} C_i \tag{24}$$

where the concentration C is calculated from the ideal gas law $C = P/RT$. This is identical to the original formulation, but representing it this way produces a much sparser Jacobian for the three mechanisms, because the majority of β_{ij}'s are set to zero. A comparison of the sparsity pattern for the original formulation, mole formulation, and augmented density formulation is given in Table 1 and Figs. 1–3 for all three mechanisms with and without the revised third-body treatment. The subsequent numerical computations were done with the revised third-body treatment to give a fair representation of the benefit of using sparse solvers.

Next we compared CPU timings for the function evaluation, the Jacobian evaluation with analytical derivatives, and LU factorizations. These three calculations constitute the majority of the computational time done by all the solvers. This is shown for all three mechanisms and all three formulations in Table 2. As shown in this table, changing the formulation does not significantly affect the CPU timing for the function evaluation. This is because the large majority of time in the function evaluation is spent

TABLE 1

Sparsity of the Chemical Kinetic Mechanisms. Dense Refers to the Original Density Formulation, and Sparse Refers to the Mole Formulation and the Augmented Density Formulation, Which Give Exactly the Same Sparsity Pattern

	GRI-Mech		Acetylene		n-heptane	
	Non-zero Elem.	Percent Sparse	Non-zero Elem.	Percent Sparse	Non-zero Elem.	Percent Sparse
Dense	2,916	0%	10,000	0%	297,025	0%
Sparse[a]	2,679	11.4%	6,567	35.6%	30,095	89.9%
Sparse[b]	1,494	50.6%	2,742	73.1%	14,154	95.3%

[a] Original CHEMKIN-II treatment of third body concentrations.
[b] Third-body concentrations calculated by Eq. 24.

calculating the reaction rates, which is the same regardless of the formulation.

The CPU timings for the Jacobian evaluation shown in Table 2 are more interesting. First, we point out that the current implementation of analytical derivatives performs poorly for the original density formulation. This is because of two reasons: the sparse forward mode analytical derivative computation becomes more efficient as the sparsity of the system increases, whereas the density formulation is totally dense. Also, the current implementation in DAEPACK employs sparse data structures, which creates additional overhead and indirect references for dense systems that are unnecessary. So, it is necessary to take advantage of sparsity to obtain more efficient Jacobian calculations. Even with the sparse formulations, mechanisms as small and dense as the GRI-Mechanism are only slightly more efficient with the current implementation of analytic Jacobians. However, the acetylene soot mechanism shows a factor of two speedup in the Jacobian evaluation, and the n-heptane mechanism has nearly a 12-fold speedup in the Jacobian calculation, which should improve the speed of the simulation substantially.

In addition to the Jacobian evaluation, Table 2 also shows the LU factorization cost for the entire Jacobian **J** in Eq. 17. The LU factorization is only shown for the augmented density formulation, although the other formulations are similar for the dense calculations, and the mole formulation LU factorization is similar for the sparse LU factorization. The sparse LU factorization is conducted using the MA48 li-

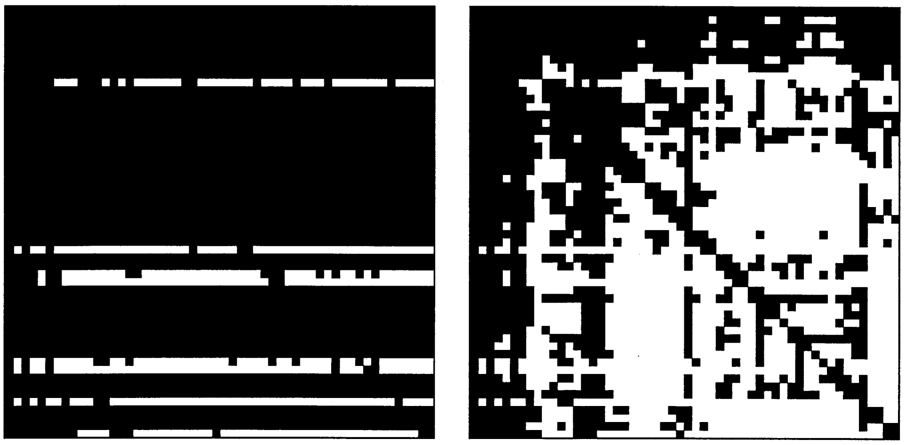

Fig. 1. Sparsity of GRI-Mech 3.0 mechanism. 54 dependent variables. With (right) and without (left) third-body sparse formulation.

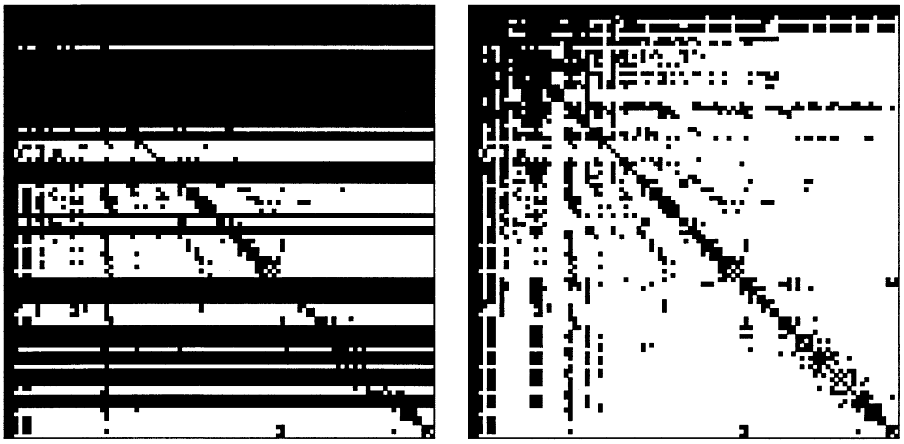

Fig. 2. Sparsity of Wang's acetylene-soot mechanism. 100 dependent variables. With (right) and without (left) third-body sparse formulation.

braries. The results here show that there is a high overhead cost to using sparse linear algebra for small, relatively dense systems. Only when we get to large, sparse systems such as the n-heptane system, can we really take advantage of sparse linear algebra, and improve the LU factorization substantially. It is interesting to note that for all cases considered in Table 2 the cost of a Jacobian evaluation is greater than the cost of a Jacobian factorization, which is somewhat contrary to the assumptions made in the design of sensitivity algorithms.

For the sensitivity calculations in SENKIN, we also examine evaluating $\partial F/\partial p$ analytically. This computation is required for all three methods discussed above. Again, we use DAEPACK generated code to do these calculations, although now the calculation of sparsity is not required. The timing results for the $\partial F/\partial p$ calculations are shown in Table 3. For the one parameter calculation, one can see that the finite difference derivative is by far the most efficient means of calculating $\partial F/\partial p$. This is because DAEPACK has been optimized to do much more complex, sparse derivatives, which requires a high overhead cost for creating the sparse structures to do the sparse forward mode analytical derivative calculation. For a single parameter, we would expect a much better result by doing a regular forward mode calcula-

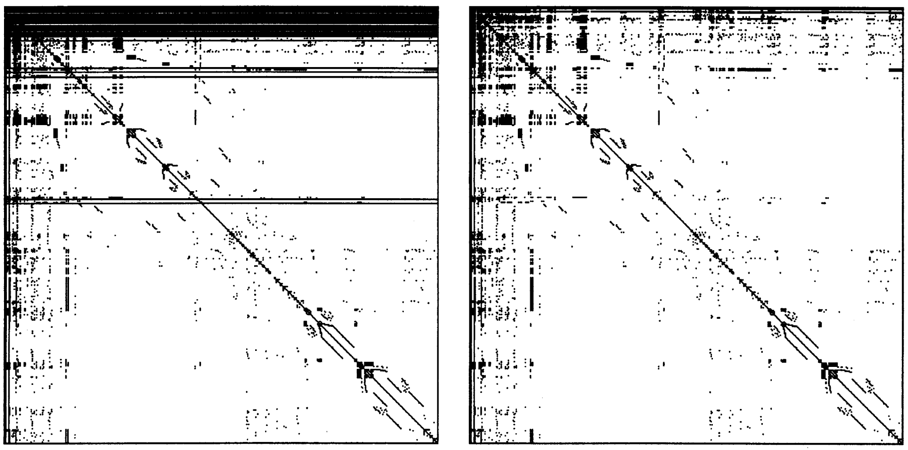

Fig. 3. Sparsity of Curran's n-heptane mechanism. 545 dependent variables. With (right) and without (left) third-body sparse formulation.

TABLE 2

CPU Timings for Function and Jacobian Evaluations, and LU Factorizations, for Each of the Three Mechanisms. Jacobians Calculated Analytically Using DAEPACK. Note that the LU Factorization is Only Shown for the Augmented Density Formulation, However all Three Formulations Behave Similarly for the Dense Solver, and the Mole and Augmented Formulations Behave Similarly for the Sparse Solver. Timings are Given in Seconds and Parenthetically as a Ratio to Function Evaluations

	GRI-Mech $N = 53, M = 325$	Acetylene-soot $N = 99, M = 498$	n-heptane $N = 544, M = 2446$
Function Evaluation			
Density	5.46×10^{-4}	8.33×10^{-4}	7.94×10^{-3}
Mole	5.23×10^{-4}	7.97×10^{-4}	7.2×10^{-3}
Aug. Density	5.51×10^{-4}	8.35×10^{-4}	8.09×10^{-3}
Jacobian Evaluation			
Original[a]	0.0300 (55)	0.0841 (101)	4.34 (546)
Density	0.063 (115)	0.16 (192)	4.15 (523)
Mole	0.0283 (54)	0.0486 (61)	0.516 (72)
Aug. Density	0.022 (41)	0.040 (48)	0.338 (43)
LU Factorization			
Dense	8.0×10^{-5}	3.6×10^{-4}	0.81
Sparse	7.8×10^{-4}	2.9×10^{-3}	0.25

[a] Density formulation and numerical derivatives.

tion [7]. As the number of parameters grows, the efficiency of the sparse analytical derivative calculation becomes apparent. For the full set of reactions, the speedup we see from using analytical derivatives ranges from nine (for the GRI-Mech) to 37 (for the n-heptane mechanism).

SIMULATION RESULTS

The next set of results (shown in Table 4 through Table 6), are the simulation results for the three mechanisms. Here we show the calculations for three different commonly used solv-

TABLE 3

CPU Timings for $\partial F/\partial p$ Calculation Using Analytical Derivatives Computed from DAEPACK. Timings are Given in Seconds and Parenthetically as a Ratio to Function Evaluations

	GRI-Mech $N = 53, M = 325$	Acetylene-soot $N = 99, M = 498$	n-heptane $N = 544, M = 2446$
Function Evaluation			
Density	5.46×10^{-4}	8.33×10^{-4}	7.94×10^{-3}
Aug. Density	5.51×10^{-4}	8.35×10^{-4}	8.09×10^{-3}
$\partial F/\partial p$ Evaluation, 1 parameter			
Numerical	0.00109 (2)	0.0017 (2)	0.0159 (2)
Density	0.0055 (10)	0.0086 (10)	0.048 (6.0)
Aug. Density	0.0055 (10)	0.0086 (10)	0.049 (6.0)
$\partial F/\partial p$ Evaluation, 100 parameters			
Numerical	0.0551 (101)	0.0841 (101)	0.802 (101)
Density	0.010 (18)	0.015 (18)	0.07 (8.8)
Aug. Density	0.010 (18)	0.015 (18)	0.07 (8.8)
$\partial F/\partial p$ Evaluation, M parameters			
Numerical	0.178 (326)	0.416 (499)	19.4 (2447)
Density	0.020 (37)	0.039 (47)	0.49 (62)
Aug. Density	0.019 (37)	0.039 (47)	0.53 (65)

TABLE 4

Comparison of Simulations for the Methane Reactor at Constant Pressure. T_a = 1,500 K, P_a = 1atm, X_{CH_4} = 0.095, X_{O2} = 0.190, X_{N_2} = 0.715. $\tau = 1 \times 10^{-3}$ sec. All Except the Original Formulation use Analytical Jacobians

						Number of		
							Failures	
Solver	Formulation	Total CPU Time (sec)	Steps	Funct. Eval.	Jac. Eval.	LU Fact.	Conv.	Error
VODE	Original	1.83	1198	2791	22	156	0	57
VODE	Density	2.84	1312	1768	26	184	0	76
VODE	Mole	1.83	1212	1625	23	156	0	57
DASSL	Density	5.77	1316	1713	74	74	0	9
DASSL	Mole	3.12	1344	1746	68	68	0	8
DASSL	Aug. density	2.75	1165	1532	78	78	0	15
DSL48S	Mole	3.12	1248	1691	70	70	0	10
DSL48S	Aug. density	2.90	1261	1707	76	76	0	16

ers, with all three formulations. We could not use the augmented density formulation with VODE because VODE is an ODE (not a DAE) solver, and we did not run the density formulation with DSL48S because the sparse linear algebra would be useless on a totally dense system. All of the calculations (except for the original calculations) use analytical Jacobians. Relative and absolute tolerances of 10^{-6} and 10^{-15} were used, respectively, for the computations. Results showed no difference between the three different formulations within the error tolerances.

There are several interesting details observed in these simulations. The first is the difference between the VODE and DASSL (and thus DSL48S) simulation strategies. VODE has a very aggressive time-step strategy, resulting in many error test failures and subsequent LU factorizations. However, by separating the LU factorization from the Jacobian evaluation, the number of Jacobian evaluations still remains relatively small. This strategy works very well for small systems (such as the GRI-Mech), where the cost of the LU factorization is much smaller than the cost of the Jacobian evaluation. However, for larger systems, such as n-heptane, this strategy becomes fairly costly.

As we would expect from Table 2, none of the improvements in analytical Jacobians or sparse linear algebra results in faster overall simulations for the GRI-Mechanism (Table 4). Only a small improvement can be seen with the larger acetylene-soot mechanism using the new tech-

TABLE 5

Comparison of Simulations for the Acetylene Reactor at Constant Pressure. T_a = 1,000 K, P_a = 1atm, $X_{C_2H_2}$ = 0.0775, X_{O_2} = 0.1938, X_{N_2} = 0.7287. $\tau = 1 \times 10^{-3}$ sec. All Except the Original Formulation use Analytical Jacobians

						Number of		
							Failures	
Solver	Formulation	Total CPU Time (sec)	Steps	Funct. Eval.	Jac. Eval.	LU Fact.	Conv.	Error
VODE	Original	6.5	1369	6063	39	207	0	77
VODE	Density	8.6	1405	2198	36	198	0	73
VODE	Mole	5.0	1482	2322	36	224	0	92
DASSL	Density	15.2	1337	1938	80	80	0	18
DASSL	Mole	5.84	1212	1839	74	74	0	17
DASSL	Aug. density	5.6	1340	1968	73	73	0	16
DSL48S	Mole	5.4	1187	1896	69	69	0	11
DSL48S	Aug. density	5.3	1376	2046	74	74	0	13

TABLE 6

Comparison of Simulations for the n-heptane Reactor at Constant Pressure. $T_a = 800$ K, $P_a = 12.5$atm, $X_{nC_7H_{16}} = 0.0014$, $X_{O_2} = 0.0252$, $X_{N_2} = 0.9734$. $\tau = 1.8$ sec. All Except the Original Formulation use Analytical Jacobians

				Number of				
							Failures	
Solver	Formulation	Total CPU Time (sec)	Steps	Funct. Eval.	Jac. Eval.	LU Fact.	Conv.	Error
VODE	Original	342	863	18302	31	99	0	20
VODE	Density	408	920	1536	33	123	0	34
VODE	Mole	279	900	1506	32	117	0	29
DASSL	Density	309	663	1156	47	47	0	6
DASSL	Mole	131	629	1124	44	44	0	7
DASSL	Aug. density	132	822	1410	45	45	0	6
DSL48S	Mole	34.8	598	1029	47	47	0	7
DSL48S	Aug. density	33.8	806	1348	55	55	0	11

niques, and most of the simulations are comparable in CPU cost. The only notable improvement for the acetylene-soot case is found with using the VODE solver and the mole formulation, because the system is still small enough to take advantage of the VODE heuristics, and the analytical Jacobian calculation is less expensive for the mole formulation.

For the larger n-heptane system (Table 6) we see substantial improvements by changing solvers, using analytical Jacobians, and exploiting sparsity. Just by switching to the DASSL solution strategies, we see a small improvement because of the reduced number of LU factorizations. By using sparse derivative formulations and the DASSL strategies, we obtain a greater speed up of around 2.5, because of the faster analytical Jacobian calculation. By taking advantage of sparse linear algebra, the speed improves by a factor of 10 over the original solver, and brings the resultant simulation time down from 6 minutes to a little over half of a minute.

From the above calculations, it is easily seen that there is no best numerical strategy for all cases. For kinetic systems, knowing the sparsity pattern of the kinetic system is essential in determining whether sparse linear algebra or dense linear algebra is most appropriate, or even (for small mechanisms) whether analytical or numerical Jacobians are more appropriate. It currently appears that only the larger, sparse kinetic systems take full advantage of analytical derivatives and sparse linear algebra, but the benefit for these systems is quite large, compared to only a small penalty for smaller systems.

SENSITIVITY RESULTS

The final set of results are for simulations with the calculation of sensitivities. The SENKIN program that comes with CHEMKIN II specifies all of the reaction rate A-factors as sensitivity parameters, saving the results at each step into an unformatted file. The SENKIN program uses the density formulation of the constant-pressure reactor, using the DASAC solver to compute the simulation and sensitivities with finite difference Jacobian and $\partial F/\partial p$ calculations. As metioned previously, we have opted to replace the DASAC solver with the DASPK and DSL48S solvers because of their greater flexibility.

Solving for the sensitivities of all reactions for a given kinetic mechanism is not always desirable, primarily because of the sheer size of the resulting data file. For instance, for the n-heptane mechanism, storing all of the sensitivity results for all of the reactions results in a file that is 10.6 MB for each time-step (assuming double precision). A typical case such as the n-heptane simulations presented in Table 6 would result in a file the size of about 8.5 gigabytes. For such large mechanisms, typically one wants to examine a specific subset of reactions, instead of the full reaction set. For this reason, in addition to running the cases for the full set of parameters (M parameters) as the original SENKIN, we also ran it for 1 parameter and 100 parameters. We have disabled writing out the sensitivity results to a file, because of the extreme file size and cost of I/O. For many of the smaller sensitivity calculations, we did compare the results with the original sensitivity calculation to ensure that the newer implementations were correct.

The results we present use two different formulations, the original dense formulation and the sparse augmented density formulation. For the density formulation, we use numerical Jacobians because this calculation is faster than the analytical Jacobian calculation for all three kinetic mechanisms using the current DAEPACK package. For the augmented density formulation, we look at using both finite difference and analytical derivatives for Jacobians and $\partial F/\partial p$ evaluations, and the effect of using sparse linear algebra instead of dense linear algebra for the solvers. Finally, we compare the three main methods for calculating sensitivities; the simultaneous corrector (Si.C.) method, the staggered corrector (St.C.) method, and the staggered direct (St.D.) method. Only the St.C. method is available for DSL48S, so we only show those results. We would like to pinpoint, for each kinetic mechanism given N_p parameters, which method, Jacobian evaluation, $\partial F/\partial p$ evaluation, and sparse/dense linear algebra provides the fastest solutions. We use DASPK to compute the simulations with dense linear algebra, and DSL48S to compute the simulations with sparse linear algebra.

The full set of results are shown in Tables 7 through 9 for the GRI-Mechanism, Tables 10 through 12 for the acetylene-soot mechanism, and Tables 13 through 15 for the n-heptane mechanism. For each of the tables, we show the effect of using the different formulations, sensitivity calculation methods, analytical versus finite difference Jacobians, and dense versus sparse linear algebra. For each of the cases, we use the same initial conditions that were used for the original simulation runs from the previous section.

Before discussing individual results, first we make some general observations about these calculations. The first observation is that the Si.C. and the St.C. methods show a large advantage when solving one parameter systems over the St.D. method, and a smaller advantage when solving 100 parameters. This is because of the much smaller number of factorizations required for these methods compared to the direct method. As the number of parameters increases, the percentage of time spent factoring the Jacobian decreases and the advantage gets washed out. For a large set of parameters, the St.D. method becomes the most efficient. This is because it uses a current factored Jacobian at every time-step, and thus can take larger time-steps than the other methods. By taking fewer total time-steps, this method does less computations for several hundred parameters, even though it factors the Jacobian at every step. These results have also been seen in previous studies [39].

Another observation is the small difference between the Si.C. and St.C. methods for many of the cases. Although the DASPK results comparing the two methods gives a slight advantage to the simultaneous corrector method, this appears to be caused by a problem with the current staggered corrector implementation in DASPK, which causes many more error test failures for the St.C. method than the Si.C. method, resulting in more time-steps and Jacobian evaluations. Comparing these values with the DSL48S St.C. implementation shows that the error test failures should more closely agree with the Si.C. method.

For all cases with one parameter, the finite difference $\partial F/\partial p$ provided the most efficient results. This is because of the poor efficiency of the analytical derivative calculation for $\partial F/\partial p$ with one parameter, shown in Table 3 and discussed above. For 100 parameters or M parameters, however, analytical $\partial F/\partial p$ has a clear advantage for most of the calculations. The reason is twofold. First, computing analytical $\partial F/\partial p$ compared to finite difference $\partial F/\partial p$ is much more efficient for large numbers of parameters, as shown in Table 3. Second, many of the cases have a considerable difference in the number of error test failures for the finite difference $\partial F/\partial p$ compared with the analytical $\partial F/\partial p$. All of the acetylene soot cases, in particular, seem to require considerably more steps

TABLE 7

GRI-Mechanism. One Parameter

Solver	Formulation	Method	Jac.	∂F/∂p	CPU Time	Steps	Jac. Eval.	Conv. Fail.	Error Fail.
DASPK	Density	Si.C.	F.D.	F.D.	9	3482	92	0	10
		St.C.			11	3413	107	0	12
		St.D.			88.5	2557	2559	0	2
	Aug. density	Si.C.	F.D.	F.D.	8.1	3106	107	0	6
				Anal.	123	3417	124	0	16
			Anal.	F.D.	7.7	3283	108	0	9
				Anal.	129	3563	105	0	11
		St.C.	F.D.	F.D.	9.8	3011	114	0	13
				Anal.	93	3020	103	0	10
			Anal.	F.D.	8.8	3126	114	0	12
				Anal.	100	3263	108	0	10
		St.D.	F.D.	F.D.	83.7	2557	2560	0	3
				Anal.	154	2557	2560	0	3
			Anal.	F.D.	63	2556	2559	0	3
				Anal.	133	2556	2559	0	3
DSL48S	Aug. density	St.C.	Anal.	F.D.	10.3	3207	114	0	17
				Anal.	93	3290	107	0	18

when using finite difference $\partial F/\partial p$ than analytical $\partial F/\partial p$.

The final general observation is that for all of the cases except the GRI-Mech and acetylene soot cases with one parameter, DSL48S with the St.C. method showed the greatest speed increase of any of the methods. This is probably due to the correct implementation of the St.C. method, as mentioned above, coupled with the benefit of sparse linear algebra for the larger cases. In many cases the advantage of DSL48S over the other methods was very slight, but for some cases it was very large (for instance, all of the n-heptane cases).

The complete timing results are given in Tables 7 through 9 for the GRI-Mechanism for 1, 100, and 325 parameters. For one parameter, we see about a 10-fold increase by using the

TABLE 8

GRI-Mechanism. One Hundred Parameters

Solver	Formulation	Method	Jac.	$\partial F/\partial p$	CPU Time	Steps	Jac. Eval.	Conv. Fail.	Error Fail.
DASPK	Density	Si.C.	F.D.	F.D.	306	3165	94	0	8
		St.C.			272	3221	109	0	12
		St.D.			306	2756	2760	0	4
	Aug. density	Si.C.	F.D.	F.D.	266	3128	114	0	16
				Anal.	198	3365	113	0	11
			Anal.	F.D.	310	3429	111	0	15
				Anal.	184	3293	100	0	16
		St.C.	F.D.	F.D.	250	3249	116	0	16
				Anal.	159	3061	109	0	9
			Anal.	F.D.	266	3474	121	0	16
				Anal.	161	3170	113	0	19
		St.D.	F.D.	F.D.	281	2767	2771	0	4
				Anal.	202	2557	2560	0	3
			Anal.	F.D.	254	2737	2742	0	5
				Anal.	181	2556	2559	0	3
DSL48S	Aug. density	St.C.	Anal.	F.D.	663	3629	97	0	7
				Anal.	163	3193	105	0	9

TABLE 9

GRI-Mechanism. Three Hundred Twenty-Five Parameters

Solver	Formulation	Method	Jac.	$\partial F/\partial p$	CPU Time	Steps	Jac. Eval.	Conv. Fail.	Error Fail.
DASPK	Density	Si.C.	F.D.	F.D.	1026	3165	94	0	8
		St.C.			1257	4235	99	0	7
		St.D.			818	2756	2760	0	4
	Aug. density	Si.C.	F.D.	F.D.	922	3128	114	0	16
				Anal.	442	3365	113	0	11
			Anal.	F.D.	1073	3429	111	0	15
				Anal.	416	3293	100	0	16
		St.C.	F.D.	F.D.	891	3345	134	0	22
				Anal.	367	3061	109	0	9
			Anal.	F.D.	922	3402	139	0	28
				Anal.	390	3277	123	0	9
		St.D.	F.D.	F.D.	767	2767	2771	0	4
				Anal.	367	2557	2560	0	3
			Anal.	F.D.	734	2737	2742	0	5
				Anal.	475	2556	2559	0	3
DSL48S	Aug. density	St.C.	Anal.	F.D.	2358	3719	101	0	16
				Anal.	377	3172	114	0	13

Si.C. or St.C. methods with finite difference Jacobian and $\partial F/\partial p$ as opposed to the St.D. method used in the original SENKIN code. For 100 parameters, no methods stand out as superior, although the St.C. method is consistently better than the Si.C. method and St.D. methods with analytical $\partial F/\partial p$, and is 1.9 times faster than the original formulation with the DASPK solver. For the full set of parameters, the largest improvement is obtained with using analytical $\partial F/\partial p$ with the staggered corrector method, although using the St.D. method with DASPK and dense linear algebra is only slightly less efficient. Dense versus sparse linear algebra has only a limited effect on the computational efficiency.

The complete timing results for the acetylene-soot model for 1, 100, and 498 parameters are given in Tables 10 through 12. With one parameter, the largest improvement in performance is again achieved by switching the method from the St.D. method to either the Si.C. or St.C. with finite difference $\partial F/\partial p$. Smaller improvements are observed with the use of analytical Jacobians, giving an overall improvement of over 26 times the original solution procedure. For 100 parameters and 498 parameters, the largest improvements are found from applying analytical $\partial F/\partial p$ rather than finite difference $\partial F/\partial p$. The most efficient method with 100 parameters is again DSL48S with analytical Jacobian and $\partial F/\partial p$, and is 3 times faster than the original solution procedure. For the full set of parameters, DASPK with the St.D. method and analytical derivatives is slightly better than DSL48S with the St.C. method, about 3.7 times faster than the original solution procedure.

Finally, the complete results are given for the n-heptane mechansims in Tables 13 through 15. This time, for one parameter, both the solution method (switching from the St.D. to the Si.C. or the St.C. methods), plus exploiting sparsity with analytical Jacobians, give signficant increases in efficiency. DSL48S with the St.C. method, analytical Jacobian, and finite difference $\partial F/\partial p$ is by far the most efficient method, and is approximately 55 times faster than the original solution procedure. The benefit of sparsity also dominates the efficiency for 100 parameters, and even 2446 parameters. For 100 parameters, DSL48S with analytical Jacobian and $\partial F/\partial p$ is about five times faster than the original formulation. For the full set of parameters, the number of steps taken by the St.C. method is much more than the St.D. method, and the cost of doing back substitutions for the additional steps decreases the benefits of using the sparse linear algebra of DSL48S and the St.C. method. We have opted to present only a subset of cases for

TABLE 10

Acetylene-Soot Mechanism. One Parameter

Solver	Formulation	Method	Jac.	$\partial F/\partial p$	CPU Time	Steps	Jac. Eval.	Conv. Fail.	Error Fail.
DASPK	Density	Si.C.	F.D.	F.D.	20	3326	121	0	20
		St.C.			24	3479	129	0	40
		St.D.			393	4150	4160	0	10
	Aug. density	Si.C.	F.D.	F.D.	23	3313	145	6	23
				Anal.	368	3427	128	0	25
			Anal.	F.D.	15	3354	126	0	25
				Anal.	222	3643	113	0	18
		St.C.	F.D.	F.D.	27	3805	152	0	69
				Anal.	179	3252	131	0	25
			Anal.	F.D.	22	3986	153	0	60
				Anal.	184	3383	134	0	34
		St.D.	F.D.	F.D.	380	4161	4173	0	12
				Anal.	314	2297	2299	0	2
			Anal.	F.D.	182	4090	4100	0	10
				Anal.	208	2297	2299	0	2
DSL48S	Aug. density	St.C.	Anal.	F.D.	16	3163	130	0	27
				Anal.	156	3208	122	0	20

the n-heptane problem with the full set of parameters. The reason for this is the clear advantage of using analytical Jacobians, analytical $\partial F/\partial p$, and sparse linear algebra. For this case, DSL48S and the St.C. method is the fastest solution procedure, but only about two times faster than the original problem solution, and only 1.4 times faster than the St.D. method using analytical Jacobians and $\partial F/\partial p$.

Like our earlier simulations without sensitivities, there is no straight forward "best" solution procedure for calculating sensitivities. It is intimately connected with the number of parameters for which one wishes to calculate sensitivities, and the general sparsity of the kinetic system. However, some clear guidelines can be given. For the larger mechanisms, one always gains a benefit by exploiting sparsity and analyt-

TABLE 11

Acetylene-Soot Mechanism. One Hundred Parameters

Solver	Formulation	Method	Jac.	$\partial F/\partial p$	CPU Time	Steps	Jac. Eval.	Conv. Fail.	Error Fail.
DASPK	Density	Si.C.	F.D.	F.D.	492	3326	121	0	20
		St.C.			520	3813	134	0	70
		St.D.			871	4150	4160	0	10
	Aug. density	Si.C.	F.D.	F.D.	502	3313	145	6	23
				Anal.	540	3403	159	9	25
			Anal.	F.D.	486	3354	126	0	25
				Anal.	383	3643	113	0	18
		St.C.	F.D.	F.D.	617	4385	133	0	63
				Anal.	323	3271	133	0	31
			Anal.	F.D.	537	4035	140	0	46
				Anal.	309	3284	137	0	39
		St.D.	F.D.	F.D.	856	4161	4173	0	12
				Anal.	404	2297	2299	0	2
			Anal.	F.D.	649	4090	4100	0	10
				Anal.	297	2297	2299	0	2
DSL48S	Aug. density	St.C.	Anal.	F.D.	911	3242	126	0	21
				Anal.	283	3139	122	0	22

TABLE 12

Acetylene-Soot Mechanism. Four Hundred Ninety Eight Parameters

Solver	Formulation	Method	Jac.	$\partial F/\partial p$	CPU Time	Steps	Jac. Eval.	Conv. Fail.	Error Fail.
DASPK	Density	Si.C.	F.D.	F.D.	2552	3326	121	0	20
		St.C.			2940	4034	176	0	78
		St.D.			2941	4150	4160	0	10
	Aug. density	DASPK-Si.C.	F.D.	F.D.	2406	3241	129	0	21
				Anal.	1237	3462	191	23	27
			Anal.	F.D.	2507	3407	124	0	26
				Anal.	1301	3643	113	0	18
		St.C.	F.D.	F.D.	2580	3757	149	0	45
				Anal.	1018	3136	137	0	25
			Anal.	F.D.	2700	3882	135	0	50
				Anal.	1036	3194	132	0	32
		St.D.	F.D.	F.D.	2882	4161	4173	0	12
				Anal.	898	2297	2299	0	2
			Anal.	F.D.	2629	4090	4100	0	10
				Anal.	787	2297	2299	0	2
DSL48S	Aug. density	St.C.	Anal.	F.D.	4903	3434	117	0	10
				Anal.	881	2967	122	0	20

ical Jacobians. For the very large mechanisms, the benefit is quite impressive, especially for small numbers of parameters. For a small number of parameters, there is no benefit to using analytical $\partial F/\partial p$ calculations; however, for medium to large numbers of parameters, analytical $\partial F/\partial p$ can help the sensitivity calculations substantially. For very large numbers of parameters, the benefit of analytical Jacobians and $\partial F/\partial p$ often gets washed out by the sheer number of back-substitutions required for the computation. In these cases, the St.D. method is the best method, because it typically requires fewer time steps than the other methods. Otherwise, the other methods typically perform as well or better than the St.D. method, and for one

TABLE 13

n-heptane Mechanism. One Parameter

Solver	Formulation	Method	Jac.	$\partial F/\partial p$	CPU Time	Steps	Jac. Eval.	Conv. Fail.	Error Fail.
DASPK	Density	Si.C.	F.D.	F.D.	667	2288	97	0	23
		St.C.			1591	2735	259	10	109
		St.D.			6007	1024	1027	0	3
	Aug. density	Si.C.	F.D.	F.D.	691	2155	98	0	23
				Anal.	1824	2182	108	0	26
			Anal.	F.D.	328	2216	95	0	27
				Anal.	1429	2154	96	0	20
		St.C.	F.D.	F.D.	1031	2267	149	0	65
				Anal.	1643	2063	114	0	31
			Anal.	F.D.	413	2076	129	0	46
				Anal.	1448	2196	155	0	59
		St.D.	F.D.	F.D.	6026	1024	1027	0	3
				Anal.	6667	1024	1027	0	3
			Anal.	F.D.	2512	1024	1027	0	3
				Anal.	2877	1024	1027	0	3
DSL48S	Aug. density	St.C.	Anal.	F.D.	108	2052	89	0	14
				Anal.	793	1984	101	0	24

TABLE 14

n-heptane Mechanism. One Hundred Parameters

Solver	Formulation	Method	Jac.	$\partial F/\partial p$	CPU Time	Steps	Jac. Eval.	Conv. Fail.	Error Fail.
DASPK	Density	Si.C.	F.D.	F.D.	6391	2288	97	0	23
		St.C.			9580	2980	291	20	116
		St.D.			8886	1024	1027	0	3
	Aug. density	Si.C.	F.D.	F.D.	5767	2155	98	0	23
				Anal.	4834	2182	108	0	26
			Anal.	F.D.	5481	2216	95	0	27
				Anal.	4437	2154	96	0	20
		St.C.	F.D.	F.D.	7507	2736	216	8	90
				Anal.	4440	2108	139	0	61
			Anal.	F.D.	5025	2176	113	0	34
				Anal.	4099	2210	145	0	52
		St.D.	F.D.	F.D.	8357	1024	1027	0	3
				Anal.	7446	1024	1027	0	3
			Anal.	F.D.	4200	1024	1027	0	3
				Anal.	3905	1024	1027	0	3
DSL48S	Aug. density	St.C.	Anal.	F.D.	4859	2141	105	0	22
				Anal.	1749	2062	94	0	14

parameter, the other methods are substantially better. These observations motivate an implementation of the staggered direct method that is able to exploit sparsity in the linear algebra.

CONCLUSIONS

This paper has discussed the application of DAEPACK to two simple codes included with CHEMKIN-II that calculate homogeneous batch reactor simulations with and without sensitivities. Using DAEPACK, we generated FORTRAN code to compute analytical Jacobians for these calculations, and also compute the sparsity pattern to use with sparse linear solvers such as the Harwell MA48 libraries. Using these codes, several different solvers and solution procedures were used with the older codes to evaluate the benefit of using these newer solution techniques.

To evaluate these new techniques and demonstrate the utility of DAEPACK, three differ-

TABLE 15

n-heptane Mechanism. Two Thousand Four Hundred Forty Six Parameters

Solver	Formulation	Method	Jac.	$\partial F/\partial p$	CPU Time	Steps	Jac. Eval.	Conv. Fail.	Error Fail.
DASPK	Density	Si.C.	F.D.	F.D.	132,880	2288	97	0	23
		St.C.			120,765	2310	159	3	64
		St.D.			47,389	1024	1027	0	3
	Aug. density	Si.C.	F.D.	F.D.	130,411	2155	98	0	23
				Anal.	91,097	2182	108	0	26
			Anal.	F.D.	130,102	2278	105	0	32
				Anal.	86,789	2154	96	0	20
		St.C.	F.D.	Anal.	72,852	2053	166	0	68
			Anal.	Anal.	75,523	2105	146	0	59
		St.D.	F.D.	F.D.	47,810	1024	1027	0	3
			Anal.	F.D.	43,714	1024	1027	0	3
				Anal.	32,248	1024	1027	0	3
DSL48S	Aug. density	St.C.	Anal.	F.D.	117,344	2029	90	0	19
				Anal.	23,123	1966	89	0	17

ent kinetic mechanisms were examined, all of which are fairly large and complex. The sparsity for each of these mechanisms was examined along with the efficiency of using the analytical Jacobians and $\partial F/\partial p$ generated using DAEPACK, and then actual simulations were performed with and without sensitivity calculations.

Results from these simulations show that no one solution procedure is most efficient for all kinetic mechanisms. Careful consideration of the sparsity and number of parameters (for sensitivity calculations) must be taken into account before determining the appropriate solution strategy. Both for simulations with and without sensitivities, we found sparsity has a very large impact for large kinetic systems, and anyone working with these types of systems could substantially reduce the amount of CPU time used by exploiting both sparsity and analytical derivatives. For smaller systems, however, the advantage is either small or nonexistent. For the constant pressure simulations conducted for this paper, we found it was also essential to do a simple reformulation of the set of dependent variables before taking advantage of sparsity.

The largest benefit of using DAEPACK generated code is it gives the modeler a great amount of flexibility in designing a solution strategy without the high cost of writing and debugging code. In particular, none of the code in the CHEMKIN-II library subroutines was altered at any time throughout this study. Using DAEPACK generated code, the analysis of sparsity for different chemical kinetic systems is very easy, and can give insight into the kinetic mechanism and help to determine the appropriate solution technique. Based on that analysis, and on the guidelines given above, a solution strategy can be found tailored to the kinetic system of interest to give the most efficient results. The interested reader may find more information on DAEPACK at the Web site http://yoric.mit.edu/daepack/daepack.html or reference [6].

This research was supported by the EPA Center on Airborne Organics at MIT and Alstom Power.

REFERENCES

1. Susnow, R. G., Dean, A. M., Green, W. H., Peczak, P., and Broadbelt, L. J. *J. Phys. Chem.* 101:3731–3740 (1997).
2. Petzold, L. R., and Zhu, W. *AIChE Journal*, 45(4):869–886 (1999).
3. Lam, S. H., and Goussis, P. A. *Int J. Chem. Kin*, 26(4):461–486 (1994).
4. Maas, U. and Pope, S. B. *24th Symposium (Intl) on Combustion*, The Combustion Institute. Pittsburgh, 1992, p. 103.
5. Peters, N. *Lecture Notes in Physics*, 241:90–109 (1985).
6. Tolsma, J. E., and Barton, P. I. *Ind. Eng. Chem. Res.*, 39(6):1826–1839 (2000).
7. Griewank, A. *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, PA, 2000.
8. Duff, I. S., and Reid, J. K. MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. Technical Report RAL-930-072, Rutherford Appleton Laboratory, October 1993.
9. Griewank, A., Juedes, D., and Utke, J. *ACM Transactions on Mathematical Software*, 22(2):131–167 (1996).
10. Shiriaev, D., Griewank, A., and Utke, J. A user guide to ADOL-F: Automatic differentiation of Fortran codes. Technical report, Institute of Scientific Computing, TU Dresden, 1995.
11. Christianson, D. Bruce. *IMA Journal of Numerical Analysis*, 12:135–150 (1992).
12. Corliss, George F. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. (Andreas Griewank and George F. Corliss, Eds.) SIAM, Philadelphia, PA, 1991, pp. 139–146.
13. Hillstrom, Kenneth E. JAKEF - A portable symbolic differentiator of functions given by algorithms. Technical Report ANL-82-48, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, ILL, 1982.
14. Horwedel, Jim E., Worley, Brian A., Oblow, E. M., and Pin, F. G. GRESS version 1.0 users manual. Technical Memorandum ORNL/TM 10835, Martin Marietta Energy Systems, Inc., Oak Ridge National Laboratory, Oak Ridge, Tenn., 1988.
15. Kubota, K. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, (Andreas Griewank and George F. Corliss, ed.), SIAM, Philadelphia, Penn., 1991. pp. 251–262.
16. Bischof, C., Carle, A., Corliss, G., Griewank, A., and Hovland, P. *Scientific Programming*, 1(1):11–29 (1992).
17. Rostaing, N., Dalmas, S., and Galligo, A. *Tellus*, 45A:558–568 (1993).
18. Giering, R., and Kaminski, T. *ACM Transactions on Mathematical Software*, 24:437–474 (1998).
19. Moore, R. E. *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979.
20. Gatzke, Edward P., Tolsma, John E., and Barton, Paul I. *Optimization and Engineering*, 2001.
21. Park, Taeshin, and Barton, Paul I. *ACM Transactions*

on Modelling and Computer Simulation, 6(2):137–165 (1996).

22. Galán, Santos, Feehery, Willian F., and Barton, Paul I. *Applied Numerical Mathematics*, 31:17–47 (1999).
23. Tolsma, John E., and Barton, Paul I. *SIAM Journal on Scientific Computing*, 2001 (in press).
24. Kee, R. J., Miller, J. A., and Jefferson, T. H. CHEMKIN: A general-purpose, problem-independent, transportable, FORTRAN chemical kinetics code package. Technical Report SAND80–8003, Sandia National Laboratories, 1980.
25. Kee, R. J., Rupley, F. M., and Miller, J. A. CHEMKIN-II: A FORTRAN chemical kinetics package for the analysis of gas-phas chemical kinetics. Technical Report SAND89–8009, Sandia National Laboratories, 1990.
26. Lutz, A. E., Kee, R. J., and Miller, J. A. SENKIN: A FORTRAN program for predicting homogeneous gas phase chemical kinetics with sensitivity analysis. Technical Report SAND87–8248 Revised, Sandia National Laboratories, 1988.
27. Smith, G. P., Golden, D. M., Frenklach, M., Moriarty, N. W., Eiteneer, B., Goldenberg, M., Bowman, C. T., Hanson, R. K., Song, S., Gardiner, Jr., W. C., Lissianski, V.V., and Qin, Z. http://www.me.berkeley.edu/gri_mech/.
28. Jessee, J.P., Gansman, R.F., and Fiveland, W.A. *Combust. Sci. Technol.* 129:113–140 (1997).
29. Bennett, B.A.V., McEnally, C.S., Pfefferle, L.D., and Smooke, M.D. *Combust. Flame*, 123:522–546 (2000).
30. Wang, H., and Frenklach, M. *Combust. Flame*, 110: 173–221 (1997).
31. Curran, H. J., Gaffuri, P., Pitz, W. J., and Westbrook, C. K. *Combust. Flame*, 114:149–177 (1998).
32. Gear, C. W. *Numerical initial value problems in ordinary differential equations.* Prentice-Hall series in automatic computation. Prentice Hall, Englewood Cliffs, NJ, 1971.
33. Brenan, K. E., Campbell, S. L., and Petzold, L. R. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* SIAM, Philadelphia, PA, 2nd edition, 1995.
34. Brown, P. N., Byrne, G. D., and Hindmarsh, A. C. *SIAM J. Sci. Stat. Comput.*, 10:1038–1051 (1989).
35. Feehery, W. F., Tolsma, J. E., and Barton, P. I. *Appl. Numer. Math.*, 25:41–54 (1997).
36. Rabitz, H., Kramer, M., and Dacol, D. *Ann. Rev. Phys. Chem.*, 34:419–461 (1983).
37. Caracotsios, M., and Stewart, W. E. *Comput. Chem. Engng.*, 9(4):359–365 (1985).
38. Maly, T., and Petzold, L. R. *Appl. Numer. Math.*, 20:57–79 (1997).
39. Li, S., Petzold, L., and Zhu, W. *Appl. Numer. Math.*, 32:161–174 (2000).

Received 11 June 2001; revised 23 October 2001; accepted 8 November 2001